

Stable Value

Rémi Forax

Oracle Developer Week - May 2025

<https://github.com/forax/stable-value>

Two questions:

Lazy initialization ?

Lazy constant initialization ?

Part I

Lazy initialization in Java ?

Why lazy init ??

In any languages

- Defer the cost until the value is needed
 - Avoid not useful computation
 - Avoid not useful storage

Strawman code (bad !)

Initialize with null + null check when the value is needed

```
public class MyClass {  
    private Database db;    // not final  
  
    public Database getDatabase() {  
        if (db != null) {  
            return db;  
        }  
        return db = new Database(...);  
    }  
}
```

Access with : `myClass.getDatabase()`

Thread safety !

The code is not thread-safe

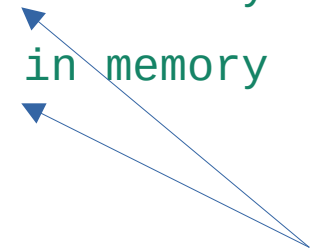
- Can create multiple Database objects !
- Can create a Database object not fully initialized !

```
public class MyClass {  
    private Database db;  
  
    public Database getDatabase() {  
        if (db != null) {  
            return db;  
        }  
        return db = new Database(...);  
    }  
}
```

Thread safe publication

In pseudo assembler

```
public class MyClass {  
    private Database db;  
  
    public Database getDatabase() {  
        if (db != null) {  
            return db;  
        }  
        var tmp = new Database    // pseudo-assembler  
        tmp.field1 = ...          // write in memory  
        tmp.field2 = ...          // write in memory  
        ...                       // write in memory  
        db = tmp;  
        return tmp;  
    }  
}
```



Can those writes be re-organized ?

TSO vs Weak model

On an Intel/AMD 64bits, stores are retired in the assembly order

- Total Store Order, stores can not be reordered

On a ARM aarch64, stores can be reordered

Java Memory Model

Allow Stores re-ordering

- Opt-in memory fences
 - Synchronized block
 - Final field
 - Volatile field

Even on a TSO CPU, JITs can reorder the stores

With a synchronized block

Make sure than the Database instance is fully initialized before being available to another thread

```
private Database db;
private Object lock = new Object();

public Database getDatabase() {
    synchronized(lock) {
        if (db != null) {
            return db;
        }
        var tmp = new db    // pseudo-assembler
        ...
        db = tmp;
        return tmp;
    }
}
```

Warning, Warning

All perf measurements have been done
on my laptop ! (MacBook Air M2)

Using JMH: <https://github.com/openjdk/jmh>

Perf – lazy init synchronized

Synchronized is a perf killer :(

Benchmark	Mode	Cnt	Score	Error	Units
LazyInitBench.lazy_synchronized_string	avgt	5	5,413	± 0,168	ns/op
LazyInitBench.string	avgt	5	0,313	± 0,001	ns/op

Double Check Locking

A design pattern from C++

Try to avoid synchronized cost by adding a null check upfront

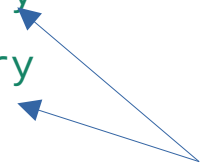
Double Check Locking (bad !)

<https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

This code has a serious bug !

```
private Database db;
private Object lock = new Object();

public static Database getDatabase() {
    if (db != null) {
        return db;
    }
    synchronized(lock) {
        if (db != null) {
            return db;
        }
        var tmp = new db      // pseudo-assembler
        tmp.field1 = ...      // write in memory
        ...
        db = tmp;             // write in memory
        return tmp;
    }
}
```



If those writes are re-organized
the object can be published without the
fields initialized

Double Check Locking (good !)

Can avoid the re-organization with volatile !

```
private volatile Database db;
private Object lock = new Object();

public Database getDatabase() {
    if (db != null) {
        return db;           // volatile read
    }
    synchronized(LOCK) {
        if (db != null) {
            return db;
        }
        var tmp = new db     // pseudo-assembler
        tmp.field1 = ...     // write
        ...
        db = tmp;           // volatile write
        return tmp;
    }
}
```

If Database is immutable (all fields are final) then volatile is not necessary

Perf – lazy init DCL

Lazy String init with a DCL is not a constant

Benchmark	Mode	Cnt	Score	Error	Units
LazyInitBench.lazy_dcl_string	avgt	5	0,731	± 0,009	ns/op
LazyInitBench.lazy_synchronized_string	avgt	5	5,413	± 0,168	ns/op
LazyInitBench.string	avgt	5	0,313	± 0,001	ns/op

Scala lazy val uses the DCL

Kotlin lazy(SYNCHRONIZED, ...) uses the DCL

Stable Value

java.lang.StableValue

Preview API in Java 25

Provide a simple API to do lazy initialization

- Should be as fast as DCL ?
- Should be as fast as a final field ?

DEMO !
(1)

Stable Value Supplier

High level API

```
private final Supplier<Database> supplier =  
    StableValue.supplier(() -> new Database(... ));  
  
public Database getDatabase() {  
    return supplier.get();  
}
```

```
Supplier<T> StableValue.supplier(Supplier<T>)
```

StableValue + .orElse()

Lower level API

```
private final StableValue<Database> value =  
    StableValue.of();  
  
public Database getDatabase() {  
    return value.orElseGet() -> new Database(...);  
}
```

StableValue.of() + .orElseGet(Supplier<T>)

Perf – stable value

Stable supplier / Stable value

Benchmark	Mode	Cnt	Score	Error	Units
LazyInitBench.lazy_dcl_string	avgt	5	0,731	± 0,009	ns/op
LazyInitBench.lazy_synchronized_string	avgt	5	5,413	± 0,168	ns/op
LazyInitBench.stable_supplier_string	avgt	5	0,894	± 0,003	ns/op
LazyInitBench.stable_value_string	avgt	5	0,829	± 0,003	ns/op
LazyInitBench.string	avgt	5	0,313	± 0,001	ns/op

StableValue now

Easier to use than the DCL

Faster than using a synchronized block

but Perf are far from the perf of a final field

- We have work to do :)

Part II

Lazy constant initialization in Java ?

Literal values ?

Literal values are constant

```
void main() {  
    System.out.println(42);  
    System.out.println("Am i a constant ?");  
}
```

```
0: getstatic      #7      // Field System.out:Ljava.io.PrintStream;  
3: bipush        42  
5: invokevirtual #13      // Method PrintStream.println:(I)V  
8: getstatic      #7      // Field System.out:Ljava.io.PrintStream;  
11: ldc          #19      // String Am i a constant ?  
13: invokevirtual #21      // Method PrintStream.println:(Ljava.lang.String;)V
```

Bytecode produces by javac (using javap)



Static final ?

Static final directly initialized are constant

```
static final int MAGIC = 40 + 2;  
static final String STRING = "Am i a constant ?";  
void main() {  
    System.out.println(MAGIC);  
    System.out.println(STRING);  
}
```

```
0: getstatic      #7      // Field System.out:Ljava.io.PrintStream;  
3: bipush        42  
5: invokevirtual #13      // Method PrintStream.println:(I)V  
8: getstatic      #7      // Field System.out:Ljava.io.PrintStream;  
11: ldc          #19      // String Am i a constant ?  
13: invokevirtual #21      // Method PrintStream.println:(Ljava.lang.String;)V
```

Constants in Java

Constants for the compiler

- Literal values (primitive + String)
- static final primitive + String
- Computations involving only constants
 - No method call

Static block ?

Static final initialized in the static block are not constant for the compiler

```
static final int MAGIC;  
static {  
    MAGIC = 42;  
}
```

```
0: getstatic #7          // Field System.out:Ljava/io/PrintStream;  
3: getstatic          #13 // Field MAGIC:I  
6: invokevirtual #19     // Method java/io/PrintStream.println:(I)V
```

Are they constant at runtime (for the JIT) ?

Perf – static init primitives

Static final initialized in the static block are constant at runtime

Benchmark	Mode	Cnt	Score	Error	Units
ConstantInStaticInitBench.magic	avgt	5	0,309	± 0,010	ns/op
ConstantInStaticInitBench.magic_block	avgt	5	0,311	± 0,005	ns/op
ConstantInStaticInitBench.string	avgt	5	0,309	± 0,004	ns/op
ConstantInStaticInitBench.string_block	avgt	5	0,310	± 0,004	ns/op

Accessing a constant object field

Is a field of a constant object a constant ?

```
class Person { final String name; ... }  
// or record Person(String name) { }  
static final Person PERSON = new Person("John");  
void main() {  
    System.out.println(PERSON.name); // constant ?  
}
```

Perf – static init objects

Fields of a constant record are constant

Benchmark	Mode	Cnt	Score	Error	Units
ConstantInStaticInitObjectBench.person	avgt	5	0,423	± 0,044	ns/op
ConstantInStaticInitObjectBench.person_record	avgt	5	0,307	± 0,005	ns/op

JEP draft: Prepare to Make Final Mean Final

<i>Author</i>	Ron Pressler & Alex Buckley
<i>Owner</i>	Ron Pressler
<i>Type</i>	Feature
<i>Scope</i>	SE
<i>Status</i>	Submitted
<i>Component</i>	core-libs
<i>Discussion</i>	jdk dash dev at openjdk dot org
<i>Reviewed by</i>	Alan Bateman, Brian Goetz
<i>Created</i>	2025/02/06 10:25
<i>Updated</i>	2025/04/26 07:43
<i>Issue</i>	8349536

Summary

Issue warnings about uses of *deep reflection* to mutate final fields. The warnings aim to prepare developers for a future release that ensures [integrity by default](#) by restricting final field mutation; this makes Java programs safer and potentially faster. Application developers can avoid both current warnings and future restrictions by selectively enabling the ability to mutate final fields where essential.

Accessing a constant List element

Is an element of a constant list a constant ?

```
static final List<String> LIST;  
static {  
    var list = new ArrayList<String>();  
    list.add("Am i a constant ?");  
    LIST = list;  
    // or LIST = List.of("Am i a constant ?");  
}  
void main() {  
    System.out.println(LIST.getFirst()); // constant ?  
}
```

Perf – static init list

Elements of a constant List.of() are constant

Benchmark	Mode	Cnt	Score	Error	Units
ConstantInStaticInitListBench.arrayList	avgt	5	0,726	± 0,006	ns/op
ConstantInStaticInitListBench.list_of	avgt	5	0,309	± 0,009	ns/op

Lazy init of constants in Java

In Java, classes are loaded lazily

- So lazy init of constants by default !

If initialization is slow or there are a lot of fields

Refactor as an afterthought

- A library API is already published
- An application uses a facade

`static final` Stable Value

DEMO !
(2)

static Stable Value Supplier

Lazy constant initialization

```
private static final Supplier<Database> SUPPLIER =  
    StableValue.supplier(() -> new Database(...));  
  
public static Database getDatabase() {  
    return SUPPLIER.get();  
}
```

Supplier<T> StableValue.supplier(Supplier<T>)

Stable Value List

`StableValue.list(size, IntFunction<E>)`

- A list of lazy initialized elements
- Elements are modifiable (`List.set`)
- List is not structurally modifiable (`List.add/remove`)

Stable Value Map

`StableValue.map(Set<K> keys, Function<K,V>)`

- A map of lazy initialized values
- Values are modifiable (`Map.put/replace`)
- Map is not structurally modifiable (`Map.put/remove`)

Perf – stable value

Lazy list and map values are/should constant

<i>Benchmark</i>	<i>Mode</i>	<i>Cnt</i>	<i>Score</i>	<i>Error</i>	<i>Units</i>
<i>ConstantStableValueBench.stable_supplier_string</i>	<i>avgt</i>	5	0,312	$\pm 0,002$	<i>ns/op</i>
<i>ConstantStableValueBench.stable_value_list</i>	<i>avgt</i>	5	0,313	$\pm 0,001$	<i>ns/op</i>
<i>ConstantStableValueBench.stable_value_map</i>	<i>avgt</i>	5	0,313	$\pm 0,001$	<i>ns/op</i>
<i>ConstantStableValueBench.stable_value_string</i>	<i>avgt</i>	5	0,313	$\pm 0,001$	<i>ns/op</i>
<i>ConstantStableValueBench.string</i>	<i>avgt</i>	5	0,313	$\pm 0,001$	<i>ns/op</i>

TLDR;

Lazy initialization in Java

Java 25 provides a new API `StableValue` (preview)

- Faster than using a synchronized
- Easier to use than the Double Check Locking
- Should behave as final after initialization (Not Yet !)
- Behave as a constant if static final

Future ??

Leyden

StableValue can be optimized inside the AOT Cache ??

Panama jextract (bridge Java <-> C)

Currently uses index instead of VarHandle because initializing a VH is slow / uses memory

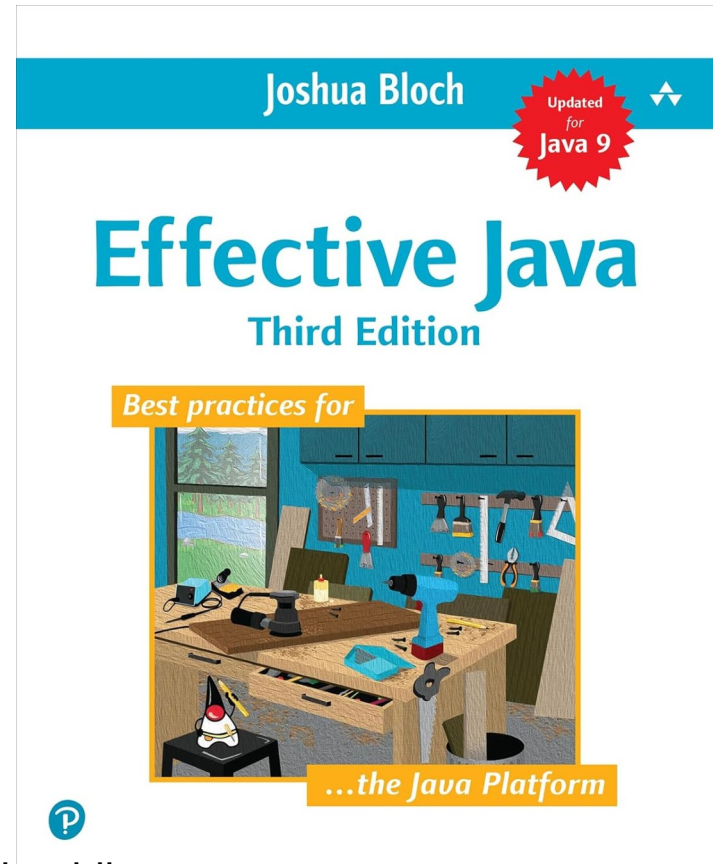
Any questions ?

Supplementary slides

Effective Java

Use lazy class init of
Java to initialize a
value lazily

=> Initialization on
demand idiom



Class holder idiom

Initialization-on-demand holder idiom

```
public static Database getDatabase() {  
    public enum Holder {  
        ;  
        static final Database DB =  
            new Database(...);  
    }  
    return Holder.DB;  
}
```

This is the Java 15+ version, with a local enum !

Perf – class holder idiom

Value of the class holder idiom is a constant

<i>// Benchmark</i>	<i>Mode</i>	<i>Cnt</i>	<i>Score</i>	<i>Error</i>	<i>Units</i>
<i>// LazyStaticInitBench.lazy_class_string</i>	<i>avgt</i>	<i>5</i>	0,312	$\pm 0,005$	<i>ns/op</i>
<i>// LazyStaticInitBench.lazy_dcl_string</i>	<i>avgt</i>	<i>5</i>	0,726	$\pm 0,009$	<i>ns/op</i>
<i>// LazyStaticInitBench.lazy_synchronized_string</i>	<i>avgt</i>	<i>5</i>	5,311	$\pm 0,042$	<i>ns/op</i>
<i>// LazyStaticInitBench.string</i>	<i>avgt</i>	<i>5</i>	0,313	$\pm 0,002$	<i>ns/op</i>

Stable Value

@Stable is not safe

Changing the value multiple times is not allowed

- But not enforced

Public API that provides safe lazy initialization patterns using @Stable

More efficient in resources than the class holder idiom

How List.of() works ?

Use an
internal
annotation
`@Stable`

```
681      @jdk.internal.ValueBased
682  ...  static final class ListN<E> extends AbstractImmutableList<E>
683      implements Serializable {
684
685          @Stable
686          private final E[] elements;
687
688          @Stable
689          private final boolean allowNulls;
690
691          // caller must ensure that elements has no nulls if allowNulls is false
692          private ListN(E[] elements, boolean allowNulls) {
693              this.elements = elements;
694              this.allowNulls = allowNulls;
695          }
696
697          @Override
698          public boolean isEmpty() {
699              return elements.length == 0;
700          }
701      }
```

@Stable

```
30  /**
31   * A field may be annotated as stable if all of its component variables
32   * changes value at most once.
33   * A field's value counts as its component value.
34   * If the field is typed as an array, then all the non-null components
35   * of the array, of depth up to the rank of the field's array type,
36   * also count as component values.
37   * By extension, any variable (either array or field) which has annotated
38   * as stable is called a stable variable, and its non-null or non-zero
39   * value is called a stable value.
40   * <p>
41   * Since all fields begin with a default value of null for references
42   * (resp., zero for primitives), it follows that this annotation indicates
43   * that the first non-null (resp., non-zero) value stored in the field
44   * will never be changed.
```